CSC3210 ASSIGNMENT-2

Attention:

An explanation is expected with every answer and may count for half of the points or more. If your answer does not have an explanation, or that explanation is incomplete, expect to lose points. This is because we want to see how you arrive at the answer.

1.) For the RISC-V assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively. (Normally, we were working with int, so we used slli x5, 2. In this assignment, we are working with long, so we use slli x5, 3 instead) **[10 POINTS]**

```
slli x30, x5, 3 // x30 = f*8
add x30, x10, x30 // x30 = &A[f]
slli x31, x6, 3 // x31 = g*8
add x31, x11, x31 // x31 = &B[g]
ld x5, 0(x30) // f = A[f]

addi x12, x30, 8
ld x30, 0(x12)
add x30, x30, x5
```

```
sd x30, 0(x31)
```

The first line (**`slli x30, x5, 3`**) explains that f is stored in x5 and 8 indicates that it's of type long–multiplying f and 8. The second line (**`add x30, x10, x30`**) stores A[f] in x30. Then, the third line (**`slli x31, x6, 3`**) shows that x6 stores g, which is multiplied by 8 because it is a long data type. The fourth line (**`add x31, x11, x31`**) stores B[g] in x31. Finally, the fifth line (**`ld x5, 0(x30)`**) loads A[f] address in x30 into the x5 register. These lines give the C code:

```
f = A[f]
```

The sixth line (**`addi x12, x30, 8`**) computes A[f+1] address. The seventh line (**`add x30, x30, x5`**) loads A[f+1] into the x30 address. The eighth line (**`add x30, x30, x5`**) computes A[f+1] + f. These lines give the C code:

```
temp = A[f+1] + f
```

The ninth line (**`lw x30, 0(x31)`**) stores the final value of temp into B[g]. Which gives the C code:

```
B[g] = temp
```

2.) Assume that registers x5 and x6 hold the values 0x8000000000000000 and 0xD000000000000000, respectively. **[6*5=30 POINTS]**

NOTE: Overflow occurs when an arithmetic operation produces a result that exceeds the representable range of the destination register.

Here we use RISC-V 64 (RV64): the size of the register is 64-bits.

a) What is the value of x30 for the following assembly code?

```
add x30, x5, x6
```

This line of code adds the values in registers x5 and x6, then stores them in x30. The value in x5 converted to hex is 1000 and x6 converted to hex is 1101, adding these two numbers gives 10101, as 15. There is a carry to the number–giving the value as 0x5000000000000000.

b) Is the result in x30 the desired result, or has there been overflow?

No, the result in x30 is not the desired result because there is an overflow. The two numbers are signed as positive because of the "0", and the two positives should result in a positive–however, the overflow carry of "1" causes this to be a negative, giving the undesired result.

c) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

```
sub x30, x5, x6
```

In order to subtract the two values, x6's binary needs to be inverted and added with 1. The inversion gives: 0x2FFFFFFFFFFFFFFF. Adding "1" to this gives

0x3000000000000000. Now, adding this new x6 value to x5 using hex is 0x8 as 1000 + 0x3 as 0011, equaling as 0xB and 1011.

d) Is the result in x30 the desired result, or has there been overflow?

Yes, the result in x30 is desired as there is no overflow. The values added are of the same sign and result in the same sign.

e) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

add x30, x5, x6

add x30, x30, x5

The value in the first statement is 0x5000000000000000, as found in part a. The value in the second line of code can be calculated as 0x5000000000000000 + 0x8000000000000000, as it is adding the result with x5. The resulting value with 0x5 as 0101 and x8 as 1000 is 0xD as 1101.

f) Is the result in x30 the desired result, or has there been overflow?

The first statement is overflowed with a positive sign and the second statement is negative, added together, they give negative as overflow.

3.) Provide the instruction type and assembly language instruction for the following binary value: 0000 0000 0001 0000 1000 0000 1011 0011two

Hint: The following figures may be helpful. **[10 POINTS]**

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (add immediate)	I	constant		reg	000	reg	0010011
lw (load word)	I	address		reg	010	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sw (store word)	S	address	reg	reg	010	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 RISC-V instruction encoding. All instructions have an opcode field, and all formats except U-type use the funct3 field. R-type instructions use the funct7 field, and immediate shifts (`slli`, `srli`, `srai`) use the funct6 field.

The instruction type for this binary value is R-type because the opcode (0110011) matches the opcode required for R-types. The funct3 code also matches with 000, and funct 7 matches with 0000000–indicating that the assembly language instruction is add.

4.) Provide the instruction type and hexadecimal representation of the following instruction:

`sw x5, 32(x30)` **[10 POINTS]**

Starting off, the `sw` instruction type indicates that this is a store word instruction and of the S-type format. Following this format, in order to get the hexadecimal code, the converted binary value is **`immed[11:5] rs2 rs1 funct3 immed[4:1,11] opcode`**. Converting this further gives: 0000001 00101 11110010 00000 0100011. Finally, converting this to hexadecimal gives: 0x025F02023.

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

5.) Assume x5 holds the value 0x0000000001010000. What is the value of x6 after the following instructions? (ORI: or Immediate) **[10 POINTS]**

NOTE: Here we use RISC-V 64 (RV64): the size of the register is 64-bits.

```
            bge x5, x0, ELSE
            jal x0, DONE
ELSE:       ori x6, x0, 2
DONE:
```

The first line of code (**`bge x5, x0, ELSE`**) means “branch of greater than or equal to” and checks if x5 is greater than x0 which is 0x0000000000000000, and moves to ELSE if true–which it is in this case. Moving from the first line (because x5 is greater than x0’s value), the code then branches to ELSE as instructed. Next, this instruction has **`ori`** which is the bitwise OR with immediate performs the action: x5 equal to x0 OR 2–which

results in x5 equaling x2 because x0 is 0. All this now gives the final answer as 0x0000000000000002.

6.) Consider a proposed new instruction named rpt. This instruction combines a loop's condition check and counter decrement into a single instruction.

For example rpt x29, loop would do the following:

```
if (x29 > 0) {
    x29 = x29 –1;
    goto loop
}
```

a) If this instruction were to be added to the RISC-V instruction set, what is the most appropriate instruction format? **[10 POINTS]**

Considering the third line of code provided (`goto loop`), the condition check, and decrement, the new instruction named rpt seems to perform loop-type operations with a conditional branching off feature. Branching is most efficiently done with B-type code as it can perform branching operations. R-type is not the most appropriate instruction format as it doesn't usually have branching functionality, I-type is also not because rpt requires checking and branching, U- and UJ-types are also not appropriate as they do not directly deal with checking, S-type is partially an appropriate instruction format as it can handle conditional checking and branching, however, SB-type is the best as it often deals with banking and branching with other instructions such as beq, bne, etc.

b) What is the shortest sequence of RISC-V instructions that performs the same operation? **[5 POINTS]**

The first line of code can be written using blt since it needs to compare x29 > 0, and jump the decrement if not:

```
blt x29, 0, DONE
```

Then, the code for decrementing can be written using addi because addi can subtract one by adding -1 as immediate:

```
addi x29, x29, -1
```

The next line of code needs to keep looping until the necessary function is complete, this can be done with jal since it jumps to the loop

```
jal x29, LOOP
```

Finally, the DONE can conclude this function:

```
DONE:
```

7.)

a) Translate the following C code to RISC-V assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the array D.

[10 POINTS]

```
for(i=0; i<a; i++)
```

```
for(j=0; j<b; j++)
      D[4*j] = i + j;
```

Registers and their values: x5 = a, x6 = b, x7 = i, x29 = j, and x10 = D's base address. The first line is the outer loop that loops i starting at 0 to a - 1 while incrementing i, in assembly, this can be done using addi to initialize i, blt to compare if x7 is lesser than x5, and addi to initialize j as 0:

```
addi x7, x0, 0
outer_loop:
   bge x7, x5, DONE
   addi x29, x0, 0
```

The inner loop and second line of code is similar to the first line, where it initializes j, has a condition, and increments until j is as large as possible while remaining less than b's value; this can be written in C with bge to end the inner loop and move on, perform the actions in the third line with add, slli, add, and sw in that respective order to compute and store i + j, and finally increment i with addi, then jal to jump as:

```
inner_loop:
   bge x29, x6, finish_inner
   add x31, x7, x29
   slli x30, x29, 2
   add x30, x10, x30
   sw x31, 0(x30)
   addi x29, x29, 1
```

```
            jal x0, inner_loop
```

Next, the code to terminate the inner loop can be written with add to increment i and jump with jal as:

```
finish_inner:
   addi x7, x7, 1
   jal x0, outer_loop
```

Finally, the program is complete with DONE:

```
DONE:
```

b) How many RISC-V instructions does it take to implement the C code from (a)? If the variables a and b are initialized to 10 and 1 and all elements of D are initially 0, what is the total number of RISC-V instructions executed to complete the loop? **[5 POINTS]**

The outer loop of code has four lines of code while the inner has 8 lines in part (a). For a and b initialized to 10 and 1 gives: 4 + 8 = 12 times 10, equaling 120 total number of RISC-V instructions executed to complete the loop.